

Chapter 1: Programming in Linux

In This Chapter

- ✓ Figuring out programming
- ✓ Exploring the software-development tools in Linux
- ✓ Compiling and linking programs with GCC
- ✓ Using make
- ✓ Debugging programs with gdb
- ✓ Understanding the implications of GNU, GPL, and LGPL

Linux comes loaded with all the tools you need to develop software. (All you have to do is install them.) In particular, it has all the GNU software-development tools, such as GCC (C and C++ compiler), GNU make, and the GNU debugger. This chapter introduces you to programming, describes the software-development tools, and shows you how to use them. Although I provide examples in the C and C++ programming languages, the focus is not on showing you how to program in those languages but on showing you how to use various software-development tools (such as compilers, make, and debugger).

The chapter concludes with a brief explanation of how the Free Software Foundation's GNU General Public License (GPL) may affect any plans you might have to develop Linux software. You need to know about the GPL because you use GNU tools and GNU libraries to develop software in Linux.

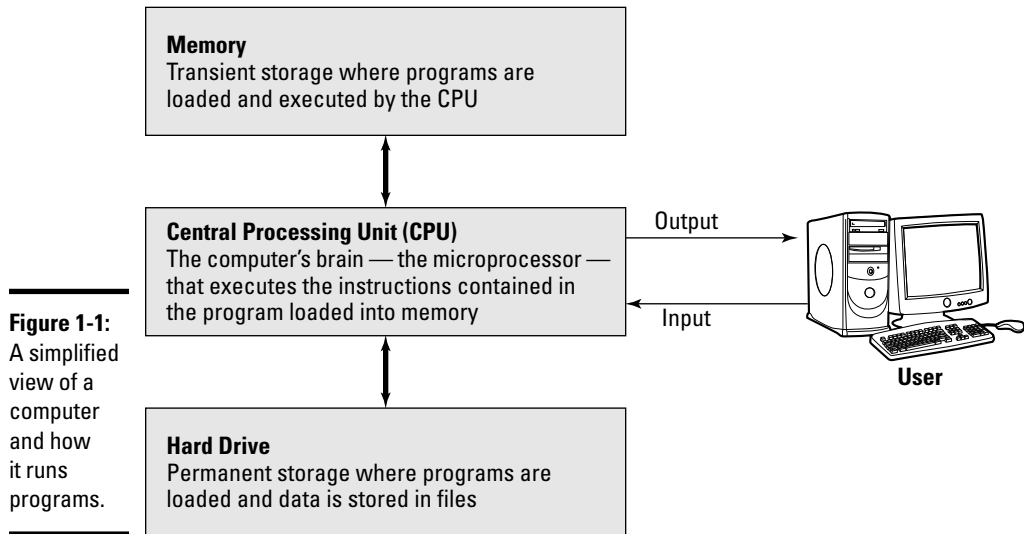
An Overview of Programming

If you've written computer programs in any programming language, you can start writing programs on your Linux system quickly. If you've never written a computer program, however, you need two basic resources before you begin to write code: a look at the basics of programming and a quick review of computers and their major parts. This section offers an overview of computer programming — just enough to get you going.

A simplified view of a computer

Before you get a feel for computer programming, you need to understand where computer programs fit in with the overall scheme of computing.

Figure 1-1 shows a simplified view of a computer, highlighting the major parts that are important to a programmer.



At the heart of a computer is the *central processing unit* (CPU) that performs the instructions contained in a computer program. The specific piece of hardware that does the job (which its makers call a *microprocessor* and the rest of us call a *chip*) varies by system: In a Pentium PC, it's a Pentium; in a Sun SPARC workstation, it's a SPARC chip; in an HP UNIX workstation, it's a PA-RISC chip. These microprocessors have different capabilities but the same mission: Tell the computer what to do.

Random Access Memory (RAM), or just *memory*, serves as the storage for computer programs while the CPU executes them. If a program works on some data, that data is also stored in memory. The contents of memory aren't permanent; they go away (never to return) when the computer is shut down or when a program is no longer running.

The *hard drive* (also referred to as the *hard disk* or *disk*) serves as the permanent storage space for computer programs and data. The hard drive is organized into files, which are in turn organized in hierarchical directories and subdirectories (somewhat like organizing paper folders into the drawers in a file cabinet). Each file is essentially a block of storage capable of holding a variety of information. For example, a file may be a human-readable text file — or it may be a collection of computer instructions that makes sense only to the CPU. When you create computer programs, you work a lot with files.

For a programmer, the other two important items are the *input* and the *output* — the way a program gets input from the user and displays output to the user. The user provides input through the keyboard and mouse, and output appears on the monitor. However, a program may also accept input from a file and send output to a file.

Role of the operating system

The *operating system* is a special collection of computer programs whose primary purpose is to load and run other programs. The operating system also acts as an interface between the software and the hardware. All operating systems include one or more command processors (called *shells* in Linux) that allow users to type commands and perform tasks, such as running a program or printing a file. Most operating systems also include a graphical user interface (such as GNOME and KDE in Linux) that allows the user to perform most tasks by clicking on-screen icons. Linux, Windows (whether the XP or Windows 7 version), and various versions of UNIX are examples of operating systems.

The operating system gives a computer its personality. For example, you can run Windows Vista or Windows XP on a PC. On that same PC, you can also install and run Linux. That means, depending on the operating system installed on it, the selfsame PC could be a Windows 7, Windows XP, or Linux system.

Computer programs are built *on top of* the operating system. That means a computer program must make use of the capabilities that the operating system includes. For example, computer programs read and write files by using the built-in capabilities of the operating system. (And if the operating system can't make coffee, no program can tell it to and still expect positive results.)

Although the details vary, most operating systems support a number of similar concepts. As a programmer, you need to be familiar with the following handful of concepts:

- ◆ A *process* is a computer program that is currently running in the computer. Most operating systems allow multiple processes to run simultaneously.
- ◆ A *command processor*, or *shell*, is a special program that allows the user to type commands and perform various tasks, such as run any program, look at a host of files, or print a file. In Windows Vista or Windows XP, you can type commands in a Command Prompt window.
- ◆ The term *command line* refers to the commands that a user types to the command processor. Usually a command line contains a command and

one or more *options* — the command is the first word in the line and the rest are the *options* (specific behaviors for the computer).

- ♦ *Environment variables* are essentially text strings with names. For example, the `PATH` environment variable refers to a string that contains the names of directories. Operating systems use environment variables to provide useful information to processes. To see a list of environment variables in a Windows system, type `set` in the Command Prompt window. In Linux, you can type `printenv` to see the environment variables.

Basics of computer programming

A *computer program* is a sequence of instructions for performing a specific task, such as adding two numbers or searching for some text in a file. Consequently, computer programming involves *creating* that list of instructions, telling the computer how to complete a specific task. The exact instructions depend on the programming language that you use. For most programming languages, you have to go through the following steps to create a computer program:

- 1. Use a text editor to type the sequence of commands from the programming language.**

This sequence of commands accomplishes your task. This human-readable version of the program is called the *source file* or *source code*. You can create the source file with any application (such as a word processor) that can save a document in plain-text form.



Always save your source code as plain text. (The filename depends on the type of programming language.) Word processors can sometimes put extra instructions in their documents that tell the computer to display the text in a particular font or other format. Saving the file as plain text deletes any and all such extra instructions. Trust me, your program is much better off without such stuff.

- 2. Use a *compiler* program to convert that text file — the source code — from human-readable form into machine-readable *object code*.**

Typically, this step also combines several object code files into a single machine-readable computer program, something that the computer can run.

- 3. Use a special program called a *debugger* to track down any errors and find which lines in the source file might have caused the errors.**
- 4. Go back to Step 1 and use the text editor to correct the errors, and repeat the rest of the steps.**

These steps are referred to as the *edit-compile-debug* cycle of programming because most programmers have to repeat this sequence several times before a program works correctly.

In addition to knowing the basic programming steps, you also need to be familiar with the following terms and concepts:

- ◆ *Variables* are used to store different types of data. You can think of each variable as being a placeholder for data — kind of like a mailbox, with a name and room to store data. The content of the variable is its value.
- ◆ *Expressions* combine variables by using operators. An expression may add several variables; another may extract a part of a string.
- ◆ *Statements* perform some action, such as assigning a value to a variable or printing a string.
- ◆ *Flow-control statements* allow statements to execute in various orders, depending on the value of some expression. Typically, flow-control statements include `for`, `do-while`, `while`, and `if-then-else` statements.
- ◆ *Functions* (also called *subroutines* or *routines*) allow you to group several statements and give the group a name. This feature allows you to execute the same set of statements by invoking the function that represents those statements. Typically, a programming language provides many predefined functions to perform tasks, such as opening (and reading from) a file.

Exploring the Software-Development Tools in Linux

Linux includes the following traditional UNIX software-development tools:

- ◆ **Text editors** such as `vi` and `emacs` for editing the source code. (To find out more about `vi`, see Book II, Chapter 6.)
- ◆ A **C compiler** for compiling and linking programs written in C — the programming language of choice for writing UNIX applications (though nowadays, many programmers are turning to C++ and Java). Linux includes the GNU C and C++ compilers. Originally, the GNU C Compiler was known as GCC — which now stands for *GNU Compiler Collection*. (See a description at <http://gcc.gnu.org>.)
- ◆ The **GNU make utility** for automating the software *build process* — the process of combining object modules into an executable or a library. (The operating system can load and run an executable; a *library* is a collection of binary code that can be used by executables.)



- ◆ A **debugger** for debugging programs. Linux includes the GNU debugger `gdb`.
- ◆ A **version-control system** to keep track of various revisions of a source file. Linux comes with RCS (Revision Control System) and CVS (Concurrent Versions System). Nowadays, most open source projects use CVS as their version control system, but a recent version control system called Subversion is being developed as a replacement for CVS.

You can install these software-development tools in any Linux distribution:

- ◆ **Xandros:** Usually, the tools are installed by default.
- ◆ **Fedora:** Select the Development Tools package during installation.
- ◆ **Debian:** Type `apt-get install gcc` and then `apt-get install libc6-dev` in a terminal window.
- ◆ **SUSE:** Choose Main Menu⇨System⇨YaST, click Software on the left side of the window, and then click Install and Remove Software. Type `gcc` in the search field in YaST, select the relevant packages from the search results, and click Accept to install. If you find any missing packages, you can install them in a similar manner.

The next few sections briefly describe how to use these software-development tools to write applications for Linux.

GNU C and C++ compilers

The most important software-development tool in Linux is GCC — the GNU C and C++ compiler. In fact, GCC can compile three languages: C, C++, and Objective-C (a language that adds object-oriented programming capabilities to C). You use the same `gcc` command to compile and link both C and C++ source files. The GCC compiler supports ANSI standard C, making it easy to port any ANSI C program to Linux. In addition, if you've ever used a C compiler on other UNIX systems, you should feel right at home with GCC.

Using GCC

Use the `gcc` command to invoke GCC. By default, when you use the `gcc` command on a source file, GCC preprocesses, compiles, and links to create an executable file. However, you can use GCC options to stop this process at an intermediate stage. For example, you might invoke `gcc` by using the `-c` option to compile a source file and to generate an object file, but not to perform the link step.

Using GCC to compile and link a few C source files is easy. Suppose you want to compile and link a simple program made up of two source files. To

accomplish this use the following program source; the task that is stored in the file `area.c` computes the area of a circle whose radius is specified at the command line:

```
#include <stdio.h>
#include <stdlib.h>
/* Function prototype */
double area_of_circle(double r);
int main(int argc, char **argv)
{
    if(argc < 2)
    {
        printf("Usage: %s radius\n", argv[0]);
        exit(1);
    }
    else
    {
        double radius = atof(argv[1]);
        double area = area_of_circle(radius);
        printf("Area of circle with radius %f = %f\n",
            radius, area);
    }
    return 0;
}
```

You need another file that actually computes the area of a circle. Here's the listing for the `circle.c` file, which defines a function that computes the area of a circle:

```
#include <math.h>
#define SQUARE(x) ((x)*(x))
double area_of_circle(double r)
{
    return 4.0 * M_PI * SQUARE(r);
}
```

For such a simple program, of course, we could place everything in a single file, but this example was contrived a bit to show you how to handle multiple files.

To compile these two files and to create an executable file named `area`, use this command:

```
gcc -o area area.c circle.c
```

This invocation of GCC uses the `-o` option to specify the name of the executable file. (If you don't specify the name of an output file with the `-o` option, GCC saves the executable code in a file named `a.out`.)

If you have too many source files to compile and link, you can compile the files individually and generate *object files* (that have the `.o` extension). That way, when you change a source file, you need to compile only that file — you just link the compiled file to all the object files. The following commands show how to separate the compile and link steps for the sample program:

```
gcc -c area.c
gcc -c circle.c
gcc -o area area.o circle.o
```

The first two commands run `gcc` with the `-c` option compiling the source files. The third `gcc` command links the object files into an executable named `area`.

Compiling C++ programs

GNU CC is a combined C and C++ compiler, so the `gcc` command also can compile C++ source files. GCC uses the file extension to determine whether a file is C or C++. C files have a lowercase `.c` extension whereas C++ files end with `.C` or `.cpp`.



Although the `gcc` command can compile a C++ file, that command doesn't automatically link with various class libraries that C++ programs typically require. Compiling and linking a C++ program by using the `g++` command is easy because it runs `gcc` with appropriate options.

Suppose that you want to compile the following simple C++ program stored in a file named `hello.C`. (Using an uppercase C extension for C++ source files is customary.)

```
#include <iostream>
int main()
{
    using namespace std;
    cout << "Hello from Linux!" << endl;
}
```

To compile and link this program into an executable program named `hello`, use this command:

```
g++ -o hello hello.C
```

The command creates the `hello` executable, which you can run as follows:

```
./hello
```

The program displays the following output:

```
Hello from Linux!
```

A host of GCC options controls various aspects of compiling C and C++ programs.

Exploring GCC options

Here's the basic syntax of the `gcc` command:

```
gcc options filenames
```

Each option starts with a hyphen (-) and usually has a long name, such as `-funsigned-char` or `-finline-functions`. Many commonly used options are short, however, such as `-c`, to compile only, and `-g`, to generate debugging information (needed to debug the program by using the GNU debugger, `gdb`).

You can view a summary of all GCC options by typing the following command in a terminal window:

```
man gcc
```

Then you can browse through the commonly used GCC options. Usually, you don't have to provide GCC options explicitly because the default settings are fine for most applications. Table 1-1 lists some of the GCC options you may use.

Table 1-1 Common GCC Options	
Option	Meaning
<code>-ansi</code>	Supports only ANSI standard C syntax. (This option disables some GNU C-specific features, such as the <code>__asm__</code> and <code>__typeof__</code> keywords.) When used with <code>g++</code> , supports only ISO standard C++.
<code>-c</code>	Compiles and generates only the object file.
<code>-DMACRO</code>	Defines the macro with the string "1" as its value.
<code>-DMACRO=DEFN</code>	Defines the macro as <code>DEFN</code> , where <code>DEFN</code> is some text string.
<code>-E</code>	Runs only the C preprocessor.
<code>-fallow-single-precision</code>	Performs all math operations in single precision.
<code>-fpcc-struct-return</code>	Returns all <code>struct</code> and <code>union</code> values in memory, rather than in registers. (Returning values this way is less efficient, but at least it's compatible with other compilers.)

(continued)

Table 1-1 (continued)

<i>Option</i>	<i>Meaning</i>
<code>-fPIC</code>	Generates position-independent code (PIC) suitable for use in a shared library.
<code>-freg-struct-return</code>	When possible, returns <code>struct</code> and <code>union</code> values registers.
<code>-g</code>	Generates debugging information. (The GNU debugger can use this information.)
<code>-I DIRECTORY</code>	Searches the specified directory for files that you include by using the <code>#include</code> preprocessor directive.
<code>-L DIRECTORY</code>	Searches the specified directory for libraries.
<code>-l LIBRARY</code>	Searches the specified library when linking.
<code>-mcpu=cputype</code>	Optimizes code for a specific processor. (<code>cputype</code> can take many different values — some common ones are <code>i386</code> , <code>i486</code> , <code>i586</code> , <code>i686</code> , <code>pentium</code> , <code>pentiumpro</code> , <code>pentium2</code> , <code>pentium3</code> , <code>pentium4</code> .)
<code>-o FILE</code>	Generates the specified output file (used to designate the name of an executable file).
<code>-O0</code> (two zeros)	Does not optimize.
<code>-O</code> or <code>-O1</code> (letter O)	Optimizes the generated code.
<code>-O2</code> (letter O)	Optimizes even more.
<code>-O3</code> (letter O)	Performs optimizations beyond those done for <code>-O2</code> .
<code>-Os</code> (letter O)	Optimizes for size (to reduce the total amount of code).
<code>-pedantic</code>	Generates errors if any non-ANSI standard extensions are used.
<code>-pg</code>	Adds extra code to the program so that, when run, this program generates information that the <code>gprof</code> program can use to display timing details for various parts of the program.
<code>-shared</code>	Generates a shared object file (typically used to create a shared library).
<code>-U MACRO</code>	Undefines the specified macros
<code>-v</code>	Displays the GCC version number.
<code>-w</code>	Doesn't generate warning messages.
<code>-W1, OPTION</code>	Passes the <code>OPTION</code> string (containing multiple comma-separated options) to the linker. To create a shared library named <code>libXXX.so.1</code> , for example, use the following flag: <code>-W1, -soname, libXXX.so.1</code> .

The GNU make utility

When an application is made up of more than a few source files, compiling and linking the files by manually typing the `gcc` command can get tiresome. Also, you don't want to compile every file whenever you change something in a single source file. These situations are where the GNU `make` utility comes to your rescue.

The `make` utility works by reading and interpreting a *makefile* — a text file that describes which files are required to build a particular program as well as how to compile and link the files to build the program. Whenever you change one or more files, `make` determines which files to recompile — and it issues the appropriate commands for compiling those files and rebuilding the program.

Makefile names

By default, GNU `make` looks for a *makefile* that has one of the following names, in the order shown:

- ◆ `GNUmakefile`
- ◆ `makefile`
- ◆ `Makefile`

In UNIX systems, using `Makefile` as the name of the *makefile* is customary because it appears near the beginning of directory listings, where uppercase names appear before lowercase names.

When you download software from the Internet, you usually find a `Makefile`, together with the source files. To build the software, you only have to type `make` at the shell prompt and `make` takes care of all the steps necessary to build the software.

If your *makefile* doesn't have a standard name (such as `Makefile`), you have to use the `-f` option with `make` to specify the *makefile* name. If your *makefile* is called `myprogram.mak`, for example, you have to run `make` using the following command line:

```
make -f myprogram.mak
```

The makefile

For a program made up of several source and header files, the *makefile* specifies the following:

- ◆ The items that `make` creates — usually the object files and the executable. Using the term *target* to refer to any item that `make` has to create is common.
- ◆ The files or other actions required to create the target.
- ◆ Which commands to execute to create each target.

Suppose that you have a C++ source file named `form.C` that contains the following preprocessor directive:

```
#include "form.h" // Include header file
```

The object file `form.o` clearly depends on the source file `form.C` and the header file `form.h`. In addition to these dependencies, you must specify how `make` converts the `form.C` file to the object file `form.o`. Suppose that you want `make` to invoke `g++` (because the source file is in C++) with these options:

- ◆ `-c` (compile only)
- ◆ `-g` (generate debugging information)
- ◆ `-O2` (optimize some)

In the `makefile`, you can express these options with the following rule:

```
# This a comment in the makefile
# The following lines indicate how form.o depends
# on form.C and form.h and how to create form.o.
form.o: form.C form.h
g++ -c -g -O2 form.C
```

In this example, the first noncomment line shows `form.o` as the target and `form.C` and `form.h` as the dependent files.



The line following the dependency indicates how to build the target from its dependents. This line must start with a tab. Otherwise, the `make` command exits with an error message, and you're left scratching your head because when you look at the `makefile` in a text editor, you can't tell the difference between a tab and a space. Now that you know the secret, the fix is to replace the spaces at the beginning of the offending line with a single tab.

The benefit of using `make` is that it prevents unnecessary compilations. After all, you can run `g++` (or `gcc`) from a shell script to compile and link all the files that make up your application, but the shell script compiles everything, even if the compilations are unnecessary. GNU `make`, on the other hand, builds a target only if one or more of its dependents have changed since the last time the target was built. `make` verifies this change by examining the time of the last modification of the target and the dependents.

`make` treats the target as the name of a goal to be achieved; the target doesn't have to be a file. You can have a rule such as this one:

```
clean:
rm -f *.o
```

This rule specifies an abstract target named `clean` that doesn't depend on anything. This dependency statement says that to create the target `clean`, GNU `make` invokes the command `rm -f *.o`, which deletes all files that have the `.o` extension (namely, the object files). Thus, the effect of creating the target named `clean` is to delete the object files.

Variables (or macros)

In addition to the basic capability of building targets from dependents, GNU `make` includes many features that make it easy for you to express the dependencies and rules for building a target from its dependents. If you need to compile a large number of C++ files by using GCC with the same options, for example, typing the options for each file is tedious. You can avoid this repetitive task by defining a variable or macro in `make` as follows:

```
# Define macros for name of compiler
CXX= g++
# Define a macro for the GCC flags
CXXFLAGS= -O2 -g -mcpu=i686
# A rule for building an object file
form.o: form.C form.h
$(CXX) -c $(CXXFLAGS) form.C
```

In this example, `CXX` and `CXXFLAGS` are `make` variables. (GNU `make` prefers to call them *variables*, but most UNIX `make` utilities call them *macros*.)

To use a variable anywhere in the `makefile`, start with a dollar sign (\$) followed by the variable within parentheses. GNU `make` replaces all occurrences of a variable with its definition; thus, it replaces all occurrences of `$(CXXFLAGS)` with the string `-O2 -g -mcpu=i686`.

GNU `make` has several predefined variables that have special meanings. Table 1-2 lists these variables. In addition to the variables listed in Table 1-2, GNU `make` considers all environment variables (such as `PATH` and `HOME`) to be predefined variables as well.

Table 1-2 Some Predefined Variables in GNU `make`

<i>Variable</i>	<i>Meaning</i>
<code>\$\$</code>	Member name for targets that are archives. If the target is <code>libDisp.a(image.o)</code> , for example, <code>\$\$</code> is <code>image.o</code> .
<code>\$*</code>	Name of the target file without the extension.
<code>+\$</code>	Names of all dependent files with duplicate dependencies, listed in their order of occurrence.
<code>\$<</code>	The name of the first dependent file.

(continued)

Table 1-2 (continued)

<i>Variable</i>	<i>Meaning</i>
\$?	Names of all dependent files (with spaces between the names) that are newer than the target.
\$@	Complete name of the target. If the target is <code>libDisp.a image.o</code> , for example, <code>\$@</code> is <code>libDisp.a</code> .
\$^	Names of all dependent files, with spaces between the names. Duplicates are removed from the dependent filenames.
AR	Name of the archive-maintaining program (default value: <code>ar</code>).
ARFLAGS	Flags for the archive-maintaining program (default value: <code>rv</code>).
AS	Name of the assembler program that converts the assembly language to object code (default value: <code>as</code>).
ASFLAGS	Flags for the assembler.
CC	Name of the C compiler (default value: <code>cc</code>).
CFLAGS	Flags that are passed to the C compiler.
CO	Name of the program that extracts a file from RCS (the default value is <code>co</code>).
COFLAGS	Flags for the RCS <code>co</code> program.
CPP	Name of the C preprocessor (default value: <code>\$ (CC) -E</code>).
CPPFLAGS	Flags for the C preprocessor.
CXX	Name of the C++ compiler (default value: <code>g++</code>).
CXXFLAGS	Flags that are passed to the C++ compiler.
FC	Name of the FORTRAN compiler (default value: <code>f77</code>).
FFLAGS	Flags for the FORTRAN compiler.
LDFLAGS	Flags for the compiler when it's supposed to invoke the linker <code>ld</code> .
RM	Name of the command to delete a file (Default value: <code>rm -f</code>).

A sample makefile

You can write a `makefile` easily if you use the predefined variables of GNU `make` and its built-in rules. Consider, for example, a `makefile` that creates the executable `xdraw` from three C source files (`xdraw.c`, `xviewobj.c`, and `shapes.c`) and two header files (`xdraw.h` and `shapes.h`). Assume that each source file includes one of the header files. Given these facts, here is what a sample `makefile` may look like:

```
#####
# Sample makefile
# Comments start with '#'
#
#####
# Use standard variables to define compile and link flags
CFLAGS= -g -O2
# Define the target "all"
all: xdraw
OBSJ=xdraw.o xviewobj.o shapes.o
xdraw: $(OBSJ)
# Object files
xdraw.o: Makefile xdraw.c xdraw.h
xviewobj.o: Makefile xviewobj.c xdraw.h
shapes.o: Makefile shapes.c shapes.h
```

This makefile relies on GNU make's implicit rules. The conversion of .c files to .o files uses the built-in rule. Defining the variable CFLAGS passes the flags to the C compiler.



The target named `all` is defined as the first target for a reason — if you run GNU make without specifying any targets in the command line (see the make syntax described in the following section), the command builds the first target it finds in the makefile. By defining the first target `all` as `xdraw`, you can ensure that make builds this executable file, even if you don't explicitly specify it as a target. UNIX programmers traditionally use `all` as the name of the first target, but the target's name is immaterial; what matters is that it's the first target in the makefile.

How to run make

Typically, you run make by simply typing the following command at the shell prompt:

```
make
```

When run this way, GNU make looks for a file named `GNUmakefile`, `makefile`, or `Makefile` — in that order. If make finds one of these makefiles, it builds the first target specified in that makefile. However, if make doesn't find an appropriate makefile, it displays the following error message and exits:

```
make: *** No targets specified and no makefile found. Stop.
```

If your makefile happens to have a different name from the default names, you have to use the `-f` option to specify the makefile. The syntax of the make command with this option is

```
make -f filename
```

where *filename* is the name of the makefile.

Even when you have a `makefile` with a default name such as `Makefile`, you may want to build a specific target out of several targets defined in the `makefile`. In that case, you have to use the following syntax when you run `make`:

```
make target
```

For example, if the `makefile` contains the target named `clean`, you can build that target with this command:

```
make clean
```

Another special syntax overrides the value of a `make` variable. For example, GNU `make` uses the `CFLAGS` variable to hold the flags used when compiling C files. You can override the value of this variable when you invoke `make`. Here's an example of how you can define `CFLAGS` as the option `-g -O2`:

```
make CFLAGS="-g -O2"
```

In addition to these options, GNU `make` accepts several other command-line options. Table 1-3 lists the GNU `make` options.

Table 1-3 Options for GNU make	
Option	Meaning
<code>-b</code>	Ignores the variable given but accepts that variable for compatibility with other versions of <code>make</code> .
<code>-C DIR</code>	Changes to the specified directory before reading the <code>makefile</code> .
<code>-d</code>	Prints debugging information.
<code>-e</code>	Allows environment variables to override definitions of similarly named variables in the <code>makefile</code> .
<code>-f FILE</code>	Reads <code>FILE</code> as the <code>makefile</code> .
<code>-h</code>	Displays the list of <code>make</code> options.
<code>-i</code>	Ignores all errors in commands executed when building a target.
<code>-I DIR</code>	Searches the specified directory for included <code>makefiles</code> . (The capability to include a file in a <code>makefile</code> is unique to GNU <code>make</code> .)
<code>-j NUM</code>	Specifies the number of commands that the <code>make</code> command can run simultaneously.
<code>-k</code>	Continues to build unrelated targets, even if an error occurs when building one of the targets.
<code>-l LOAD</code>	Doesn't start a new job if load average is at least <code>LOAD</code> (a floating-point number).

<i>Option</i>	<i>Meaning</i>
-m	Ignores the variable given but accepts that variable for compatibility with other versions of <code>make</code> .
-n	Prints the commands to execute but does not execute them.
-o <i>FILE</i>	Does not rebuild the file named <i>FILE</i> , even if it is older than its dependents.
-p	Displays the <code>make</code> database of variables and implicit rules.
-q	Does not run anything, but returns 0 (zero) if all targets are up to date, 1 if anything needs updating, or 2 if an error occurs.
-r	Gets rid of all built-in rules.
-R	Gets rid of all built-in variables and rules.
-s	Works silently (without displaying the commands as they execute).
-t	Changes the timestamp of the files.
-v	Displays the version number of <code>make</code> and a copyright notice.
-w	Displays the name of the working directory before and after processing the <code>makefile</code> .
-W <i>FILE</i>	Assumes that the specified file has been modified (used with <code>-n</code> to see what happens if you modify that file).

The GNU debugger

Although `make` automates the process of building a program, that part of programming is the least of your worries when a program doesn't work correctly or when a program suddenly quits with an error message. You need a debugger to find the cause of program errors. Linux includes `gdb` — the versatile GNU debugger with a command-line interface.

Like any debugger, `gdb` lets you perform typical debugging tasks, such as the following:

- ◆ Set a breakpoint so that the program stops at a specified line
- ◆ Watch the values of variables in the program
- ◆ Step through the program one line at a time
- ◆ Change variables in an attempt to correct errors

The `gdb` debugger can debug C and C++ programs.

Preparing to debug a program

If you want to debug a program by using `gdb`, you have to ensure that the compiler generates and places debugging information in the executable. The debugging information contains the names of variables in your program and the mapping of addresses in the executable file to lines of code in the source file. `gdb` needs this information to perform its functions, such as stopping after executing a specified line of source code.



To make sure that the executable is properly prepared for debugging, use the `-g` option with GCC. You can do this task by defining the variable `CFLAGS` in the makefile as

```
CFLAGS= -g
```

Running gdb

The most common way to debug a program is to run `gdb` by using the following command:

```
gdb progname
```

`progname` is the name of the program's executable file. After `progname` runs, `gdb` displays the following message and prompts you for a command:

```
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i586-suse-linux".
(gdb)
```

You can type `gdb` commands at the `(gdb)` prompt. One useful command, `help`, displays a list of commands, as the next listing shows:

```
(gdb) help
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands
Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

To quit `gdb`, type `q` and then press Enter.

`gdb` has a large number of commands, but you need only a few to find the cause of an error quickly. Table 1-4 lists the commonly used `gdb` commands.

Table 1-4	Common <code>gdb</code> Commands
<i>This Command</i>	<i>Does the Following</i>
<code>break NUM</code>	Sets a breakpoint at the specified line number, <i>NUM</i> . (The debugger stops at breakpoints.)
<code>bt</code>	Displays a trace of all stack frames. (This command shows you the sequence of function calls so far.)
<code>clear FILENAME: NUM</code>	Deletes the breakpoint at a specific line number, <i>NUM</i> , in the source file <i>FILENAME</i> . For example, <code>clear xdraw.c:8</code> clears the breakpoint at line 8 of file <code>xdraw.c</code> .
<code>continue</code>	Continues running the program being debugged. (Use this command after the program stops due to a signal or breakpoint.)
<code>display EXPR</code>	Displays the value of an expression, <i>EXPR</i> (consisting of variables defined in the program) each time the program stops.
<code>file FILE</code>	Loads the specified executable file, <i>FILE</i> , for debugging.
<code>help NAME</code>	Displays help on the command named <i>NAME</i> .
<code>info break</code>	Displays a list of current breakpoints, including information on how many times each breakpoint is reached.
<code>info files</code>	Displays detailed information about the file being debugged.
<code>info func</code>	Displays all function names.
<code>info local</code>	Displays information about local variables of the current function.
<code>info prog</code>	Displays the execution status of the program being debugged.

(continued)

Table 1-4 (continued)

<code>info var</code>	Displays all global and static variable names.
<code>kill</code>	Ends the program you're debugging.
<code>list</code>	Lists a section of the source code.
<code>make</code>	Runs the <code>make</code> utility to rebuild the executable without leaving <code>gdb</code> .
<code>next</code>	Advances one line of source code in the current function without stepping into other functions.
<code>print EXPR</code>	Shows the value of the expression <code>EXPR</code> .
<code>quit</code>	Quits <code>gdb</code> .
<code>run</code>	Starts running the executable that is currently running.
<code>set variable VAR=VALUE</code>	Sets the value of the variable <code>VAR</code> to <code>VALUE</code> .
<code>shell CMD</code>	Executes the UNIX command <code>CMD</code> , without leaving <code>gdb</code> .
<code>step</code>	Advances one line in the current function, stepping into other functions, if any.
<code>watch VAR</code>	Shows the value of the variable named <code>VAR</code> whenever the value changes.
<code>where</code>	Displays the call sequence. Use this command to locate where your program died.
<code>x/F ADDR</code>	Examines the contents of the memory location at address <code>ADDR</code> in the format specified by the letter <code>F</code> , which can be <code>o</code> (octal), <code>x</code> (hex), <code>d</code> (decimal), <code>u</code> (unsigned decimal), <code>t</code> (binary), <code>f</code> (float), <code>a</code> (address), <code>i</code> (instruction), <code>c</code> (char), or <code>s</code> (string). You can append a letter indicating the size of data type to the format letter. Size letters are <code>b</code> (byte), <code>h</code> (halfword, 2 bytes), <code>w</code> (word, 4 bytes), and <code>g</code> (giant, 8 bytes). Typically, <code>ADDR</code> is the name of a variable or pointer.

Finding bugs by using `gdb`

To understand how you can find bugs by using `gdb`, you need to see an example. The procedure is easiest to show with a simple example, so the following, `dbgtst.c`, is a contrived program that contains a typical bug.

```
#include <stdio.h>
static char buf[256];
void read_input(char *s);
int main(void)
{
    char *input = NULL; /* Just a pointer, no storage for string */
    read_input(input);
    /* Process command. */
    printf("You typed: %s\n", input);
    /* . . . */
    return 0;
}
void read_input(char *s)
{
    printf("Command: ");
    gets(s);
}
```

This program's main function calls the `read_input` function to get a line of input from the user. The `read_input` function expects a character array in which it returns what the user types. In this example, however, `main` calls `read_input` with an uninitialized pointer — that's the bug in this simple program.

Build the program by using `gcc` with the `-g` option:

```
gcc -g -o dbgtst dbgtst.c
```

Ignore the warning message about the `gets` function being dangerous; I'm trying to use the shortcoming of that function to show how you can use `gdb` to track down errors.

To see the problem with this program, run it and type `test` at the `Command:` prompt:

```
./dbgtst
Command: test
Segmentation fault
```

The program dies after displaying the `Segmentation fault` message. For such a small program as this one, you can probably find the cause by examining the source code. In a real-world application, however, you may not immediately know what causes the error. That's when you have to use `gdb` to find the cause of the problem.

To use `gdb` to locate a bug, follow these steps:

1. Load the program under `gdb`.

For example, type `gdb dbgtst` to load a program named `dbgtst` in `gdb`.

2. Start executing the program under `gdb` by typing the `run` command. When the program prompts for input, type some input text.

The program fails as it did previously. Here's what happens with the `dbgtst` program:

```
(gdb) run
Starting program: /home/edulane/sdev/dbgtst
Command: test
Program received signal SIGSEGV, Segmentation fault.
0x400802b6 in gets () from /lib/tls/libc.so.6
(gdb)
```

3. Use the `where` command to determine where the program died.

For the `dbgtst` program, this command yields this output:

```
(gdb) where
#0 0x400802b6 in gets () from /lib/tls/libc.so.6
#1 0x08048474 in read_input (s=0x0) at dbgtst.c:16
#2 0x08048436 in main () at dbgtst.c:7
(gdb)
```

The output shows the sequence of function calls. Function call #0 — the most recent one — is to the `gets` C library function. The `gets` call originates in the `read_input` function (at line 16 of the file `dbgtst.c`), which in turn is called from the `main` function at line 7 of the `dbgtst.c` file.

4. Use the `list` command to inspect the lines of suspect source code.

In `dbgtst`, you may start with line 16 of `dbgtst.c` file, as follows:

```
(gdb) list dbgtst.c:16
11 return 0;
12 }
13 void read_input(char *s)
14 {
15     printf("Command: ");
16     gets(s);
17 }
18
(gdb)
```

After looking at this listing, you can tell that the problem may be the way `read_input` is called. Then you list the lines around line 7 in `dbgtst.c` (where the `read_input` call originates):

```
(gdb) list dbgtst.c:7
2 static char buf[256];
3 void read_input(char *s);
4 int main(void)
5 {
6     char *input = NULL; /* Just a pointer, no storage for string */
7     read_input(input);
8     /* Process command. */
9     printf("You typed: %s\n", input);
10    /* . . . */
11    return 0;
(gdb)
```

At this point, you can narrow the problem to the variable named `input`. That variable is an array, not a `NULL` (which means zero) pointer.

Fixing bugs in gdb

Sometimes you can fix a bug directly in `gdb`. For the example program in the preceding section, you can try this fix immediately after the program dies after displaying an error message. An extra buffer named `buf` is defined in the `dbgtst` program, as follows:

```
static char buf[256];
```

We can fix the problem of the uninitialized pointer by setting the variable `input` to `buf`. The following session with `gdb` corrects the problem of the uninitialized pointer. (This example picks up immediately after the program runs and dies, due to the segmentation fault.)

```
(gdb) file dbgtst
A program is being debugged already. Kill it? (y or n) y
Load new symbol table from "/home/edulaney/sw/dbgtst"? (y or n) y
Reading symbols from /home/edulaney/sw/dbgtst . . . done.
(gdb) list
1 #include <stdio.h>
2 static char buf[256];
3 void read_input(char *s);
4 int main(void)
5 {
6 char *input = NULL; /* Just a pointer, no storage for string */
7 read_input(input);
8 /* Process command. */
9 printf("You typed: %s\n", input);
10 /* . . . */
(gdb) break 7
Breakpoint 2 at 0x804842b: file dbgtst.c, line 7.
(gdb) run
Starting program: /home/edulaney/sw/dbgtst
Breakpoint 1, main () at dbgtst.c:7
7 read_input(input);
(gdb) set var input=buf
(gdb) cont
Continuing.
Command: test
You typed: test
Program exited normally.
(gdb)q
```

As the preceding listing shows, if the program is stopped just before `read_input` is called and the variable named `input` is set to `buf` (which is a valid array of characters), the rest of the program runs fine.

After finding a fix that works in `gdb`, you can make the necessary changes to the source files and make the fix permanent.

Understanding the Implications of GNU Licenses

You have to pay a price for the bounty of Linux. To protect its developers and users, Linux is distributed under the GNU GPL (General Public License), which stipulates the distribution of the source code.

The GPL doesn't mean, however, that you can't write commercial software for Linux that you want to distribute (either for free or for a price) in binary form only. You can follow all the rules and still sell your Linux applications in binary form.

When writing applications for Linux, be aware of two licenses:

- ◆ The **GNU General Public License** (GPL), which governs many Linux programs, including the Linux kernel and GCC
- ◆ The **GNU Library General Public License** (LGPL), which covers many Linux libraries



The following sections provide an overview of these licenses and some suggestions on how to meet their requirements. Don't take anything in this book as legal advice. Instead, you should read the full text for these licenses in the text files on your Linux system, and then show these licenses to your legal counsel for a full interpretation and an assessment of applicability to your business.

The GNU General Public License

The text of the GNU General Public License (GPL) is in a file named `COPYING` in various directories in your Linux system. For example, type the following command to find a copy of that file in your Linux system:

```
find /usr -name "COPYING" -print
```

After you find the file, you can change to that directory and type `more COPYING` to read the GPL. If you can't find the `COPYING` file, turn to the back of this book to read the GPL.

The GPL has nothing to do with whether you charge for the software or distribute it for free; its thrust is to keep the software free for all users. GPL requires that the software is distributed in source-code form and by stipulating that any user can copy and distribute the software in source-code form to anyone else. In addition, everyone is reminded that the software comes with absolutely no warranty.

The software that the GPL covers isn't in the public domain. Software covered by GPL is always copyrighted, and the GPL spells out the restrictions on the software's copying and distribution. From a user's point of view, of course, GPL's restrictions aren't really restrictions; the restrictions are benefits because the user is guaranteed access to the source code.



If your application uses parts of any software the GPL covers, your application is considered a *derived work*, which means that your application is also covered by the GPL and you must distribute the source code to your application.

Although the GPL covers the Linux kernel, the GPL doesn't cover your applications that use the kernel services through system calls. Those applications are considered normal use of the kernel.

If you plan to distribute your application in binary form (as most commercial software is distributed), you must make sure that your application doesn't use any parts of any software the GPL covers. Your application may end up using parts of other software when it calls functions in a library. Most libraries, however, are covered by a different GNU license, which is described in the next section.

You have to watch out for only a few library and utility programs the GPL covers. The GNU `dbm` (`gdbm`) database library is one of the prominent libraries GPL covers. The GNU `bison` parser-generator tool is another utility the GPL covers. If you allow `bison` to generate code, the GPL covers that code.



Other alternatives for the GNU `dbm` and GNU `bison` aren't covered by GPL. For a database library, you can use the Berkeley database library `db` in place of `gdbm`. For a parser-generator, you may use `yacc` instead of `bison`.

The GNU Lesser General Public License

The text of the GNU Lesser General Public License (LGPL) is in a file named `COPYING.LIB`. If you have the kernel source installed, a copy of `COPYING.LIB` file is in one of the source directories. To locate a copy of the `COPYING.LIB` file on your Linux system, type the following command in a terminal window:

```
find /usr -name "COPYING*" -print
```

This command lists all occurrences of `COPYING` and `COPYING.LIB` in your system. The `COPYING` file contains the GPL, whereas `COPYING.LIB` has the LGPL.

The LGPL is intended to allow use of libraries in your applications, even if you don't distribute source code for your application. The LGPL stipulates, however, that users must have access to the source code of the library you use and that users can make use of modified versions of those libraries.

The LGPL covers most Linux libraries, including the C library (`libc.a`). Thus, when you build your application on Linux by using the GCC compiler, your application links with code from one or more libraries that the LGPL covers. If you want to distribute your application in only binary form, you need to pay attention to LGPL.



One way to meet the intent of the LGPL is to provide the object code for your application and a `makefile` that relinks your object files with any updated Linux libraries the LGPL covers.



A better way to satisfy the LGPL is to use *dynamic linking*, in which your application and the library are separate entities, even though your application calls functions in the library when it runs. With dynamic linking, users immediately get the benefit of any updates to the libraries without ever having to relink the application.